Automatic Generation of Staged Geometric Predicates

Aleksandar Nanevski, Guy Blelloch and Robert Harper

PSCICO project
http://www.cs.cmu.edu/~pscico
Geometric Predicates

• Programs need to test relative positions of points based on their coordinates.

• Simple examples (in 2D):

  **Orientation test** (in convex hull)
  does C lie left/right/on the line AB?

  **Incircle test** (in Delaunay triangulation)
  does D lie in/out/on the circle ABC?
Geometric Predicates (cont’d)

• Orientation(A, B, C) = \text{sign} \begin{vmatrix} A_x - C_x & B_x - C_x \\ A_y - C_y & B_y - C_y \end{vmatrix}

• We only consider the \text{sign} of an expression involving +, -, and £
Convex Hull Miscomputed

Using Orientation Predicate in exact arithmetic

Using Orientation Predicate in floating-point arithmetic

!! Floating point arithmetic errs inconsistently !!
Floating-point Filters

• Get correct sign (-1, 0 or 1) of an exact expression E using floating-point!

“filters out” the easy cases

\[
\begin{align*}
\text{let } F &= E(X) \text{ in floating point} \\
\text{if } F &> \text{ error bound} \text{ then 1 else} \\
\text{if } -F &> \text{ error bound} \text{ then } -1 \text{ else} \\
\text{increase precision and repeat} \\
\text{or switch to exact arithmetic}
\end{align*}
\]

• If the correct result is 0, must go to exact phase
Current Methods

- Interval arithmetic (R.E. Moore, U. Kulisch)
  - little modification to the source program
  - does not perform well in low degrees

- Single-phase filters (Fortune & Van Wyk, LEDA)
  - compiler available
  - surpassed by multi-phase filters
Current Methods (cont’d)

• Multi-phase filters (J. Shewchuk)
  • four phases of increasing precision
  + re-use of results from earlier phases
  + between 8-35% slower than fp implementation
  - requires estimating rounding errors
  - involved and hard to implement: Shewchuk’s Insphere C implementation >500 lines of code
  - only existing are 2D and 3D Orientation and Insphere predicates, but we need more
Contributions

• Source language
  • expression involving +, -, £
  • arithmetic to be perceived as exact
  • nested anonymous functions (staging)

• Compiler to ML (or C)
  • multi-phase target code
  • error bounds estimated automatically
  • formally specified (so, can argue correctness)
Staging

• To test a set of points for position with respect to line AC:
  - compute as much as possible knowing only A and C
  - return the remaining work as function
  - apply this function to each of the points
Overview

• Introduction

• Arbitrary precision arithmetic and computing in phases

• Stage compilation

• Performance

• Conclusions and Future Work
# Notation

- **Operations:**

<table>
<thead>
<tr>
<th>Exact</th>
<th>Floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>©</td>
</tr>
<tr>
<td>-</td>
<td>a</td>
</tr>
<tr>
<td>£</td>
<td>-</td>
</tr>
</tbody>
</table>

- **Precision** \( p = \# \text{bits in mantissa} \)

- **Machine Epsilon** \( \varepsilon = 2^{-p} \)
  
  - if operation result is 1.0, then the rounding error is smaller than \( \varepsilon \)
Error recovery

Theorem. (Knuth)

If \( x = a \odot b \), the rounding error of \( x \) can be recovered by floating-point operations.

If \( \Delta \) is the rounding error, and \( c = x^a a \) then

\[
\Delta = ((a \ominus (x \ominus c)) \oplus (b \ominus c))
\]

So \( a + b = x + \Delta \) exactly.
How to Increase Precision?

- Knuth’s theorem: Pair \((x, \Delta)\) twice the precision.
- (D. Priest) Sorted list of fp-numbers arbitrary precision.

Expansion = list of magnitude decreasing non-overlapping floats

1.0101062^{120} + 1.1010062^{67} = 1.0011162^{10}

1010100000 0...0 1101000000 00............00 1001110000
Phases and re-use

- (J. Shewchuk) Arbitrary precision arithmetic phases can reuse the results of their predecessors.

Example:

\[ E = (a_1 - b_1)^2 - (a_2 - b_2)^2 \]

Let \( a_1 - b_1 = x_1 + \Delta_1 \) and \( a_2 - b_2 = x_2 + \Delta_2 \). Expand \( E \) as

\[ E = (x_1^2 - x_2^2) + (2x_1\Delta_1 - 2x_2\Delta_2) + (\Delta_1^2 - \Delta_2^2) \]

\[ \frac{O(1)}{O(\epsilon)} \frac{O(\epsilon)}{O(\epsilon^2)} \]
Phases and re-use (cont’d)

- Strategy for finding the sign of

\[
E = \frac{x_1^2 - x_2^2}{O(1)} + \frac{2x_1 \Delta_1 - 2x_2 \Delta_2}{O(\epsilon)} + \frac{\Delta_1^2 - \Delta_2^2}{O(\epsilon^2)}
\]

- Evaluate \( E \) in phases, increasing precision on demand.

- Example:

\[
\begin{align*}
A &= (x_1 \otimes x_1) \oplus (x_2 \otimes x_2) \\  & \quad \{ \text{floating point phase} \} \\
B &= x_1^2 - x_2^2 \\
C &= \text{round}(B) \oplus ((2x_1 \otimes \Delta_1) \oplus (2x_2 \otimes \Delta_2)) \\
D &= B + (2x_1 \Delta_1 - 2x_2 \Delta_2) + (\Delta_1^2 - \Delta_2^2) \\  & \quad \{ \text{exact phase} \}
\end{align*}
\]
Reusing results

\[ E = (a_1 - b_1)^2 - (a_2 - b_2)^2 \]
\[ = (x_1^2 - x_2^2) \]
\[ + (2x_1\Delta_1 - 2x_2\Delta_2) \]
\[ + (\Delta_1^2 - \Delta_2^2) \]

\[ A = (x_1 \otimes x_1) \ominus (x_2 \otimes x_2) \]
Reusing results

\[ E = (a_1 - b_1)^2 - (a_2 - b_2)^2 \]
\[ = (x_1^2 - x_2^2) \]
\[ + (2x_1 \Delta_1 - 2x_2 \Delta_2) \]
\[ + (\Delta_1^2 - \Delta_2^2) \]

\[ A = (x_1 \otimes x_1) \ominus (x_2 \otimes x_2) \]
\[ B = x_1^2 - x_2^2 \]
Reusing results

\[ E = (a_1 - b_1)^2 - (a_2 - b_2)^2 \]
\[ = (x_1^2 - x_2^2) \]
\[ + (2x_1 \Delta_1 - 2x_2 \Delta_2) \]
\[ + (\Delta_1^2 - \Delta_2^2) \]

\[ A = (x_1 \otimes x_1) \oplus (x_2 \otimes x_2) \]
\[ B = x_1^2 - x_2^2 \]
\[ C = \text{round}(B) \oplus \]
\[ ((2x_1 \otimes \Delta_1) \oplus (2x_2 \otimes \Delta_2)) \]
Reusing results

\[ E = (a_1 - b_1)^2 - (a_2 - b_2)^2 \]
\[ = (x_1^2 - x_2^2) \]
\[ + (2x_1 \Delta_1 - 2x_2 \Delta_2) \]
\[ + (\Delta_1^2 - \Delta_2^2) \]

\[ A = (x_1 \otimes x_1) \oplus (x_2 \otimes x_2) \]
\[ B = x_1^2 - x_2^2 \]
\[ C = \text{round}(B) \oplus ((2x_1 \otimes \Delta_1) \oplus (2x_2 \otimes \Delta_2)) \]
\[ D = B + \]
\[ (2x_1 \Delta_1 - 2x_2 \Delta_2) + \]
\[ (\Delta_1^2 - \Delta_2^2) \]
Overview

- Introduction
- Arbitrary precision arithmetic and computing in phases
- Stage compilation
- Performance
- Conclusion and Future Work
Source Language

• Basic arithmetic operations: +, -, £, unary -, sq
• Assignments: val x = some expression
• Nested anonymous functions
• Example:
  ```
  fn [a, b] =>
  let val ab = a + b
  val ab2 = sq ab

  fn [c] =>
  let val d = c - ab2
  end
  end
  ```
• Implicit sign test at the last assignment.
Stage compilation

• Every stage compiled into:
  - floating-point code (phase A)
  - code for estimating the rounding error

• Later phases (B, C and D) “suspended” until explicitly executed
Stage 1

• Helper code:
  - `susp : (unit -> 'a) -> (unit -> 'a)`
  - Modules for arbitrary precision arithmetic

• Compiler output:

```plaintext
fn [a, b] =>
let val ab = a + b
  val ab2 = ab * ab

val suspB = susp (fn () =>
  let val abB2 = Q.sq(ab)
  in
    abB2
  end)

val suspC = susp (fn () =>
  let val abC = ...
    val abC2 = ...
    in (abC, abC2) end)

val suspD = susp (fn () =>
  let val (abC, _) = suspC()
    val abD2 = ...
    in abD2 end)
```

```plaintext
fn [c] =>
let val d = c - ab2
end
end
```
fn \([c] \Rightarrow\)
let val d = c - ab2
  val dP = \text{abs}(c) + ab2
  val errA = 3.33067E^{-16} \times dP
in
  if d > errA then 1
  else if \text{not} d > errA then \text{not} 1 else

let val abB2 = \text{suspB}()
  ... in
  if yB > errB then 1
  else if \text{not} yB > errB then \text{not} 1 else

C
let val (_, abC2) = \text{suspC}()
  ... in
  if yC > errC then 1
  else if \text{not} yC > errC then \text{not} 1 else

D
let val abD2 = \text{suspD}()
  in
  let X = X.sign(X + (dB, X.\text{not} abD2))
end
end (* main let *)
Overview

• Introduction
• Arbitrary precision arithmetic and computing in phases
• Stage compilation
• Performance
• Conclusion and Future Work
Performance

- Experiments to match Shewchuk’s hand-generated predicates
- Target language C
- No staging!

<table>
<thead>
<tr>
<th>Uniform Random Point Distribution</th>
<th>Shewchuk’s version</th>
<th>Automatically generated version</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orient2D</td>
<td>0.208 ms</td>
<td>0.249 ms</td>
<td>1.20</td>
</tr>
<tr>
<td>Orient3D</td>
<td>0.707 ms</td>
<td>0.772 ms</td>
<td>1.09</td>
</tr>
<tr>
<td>InCircle</td>
<td>6.440 ms</td>
<td>5.600 ms</td>
<td>0.87</td>
</tr>
<tr>
<td>InSphere</td>
<td>16.430 ms</td>
<td>39.220 ms</td>
<td>2.39</td>
</tr>
</tbody>
</table>
Performance (cont’d)

<table>
<thead>
<tr>
<th>2D Delaunay Triangulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shewchuk’s version</td>
</tr>
<tr>
<td>uniform random</td>
</tr>
<tr>
<td>tilted grid</td>
</tr>
<tr>
<td>co-circular</td>
</tr>
</tbody>
</table>

- Between 1 and 2.4 times slower than hand-generated code
- Possibility for improvement with a better translation to C
Future Work

• Extend the language:
  - multiplication by $2^n$ doesn’t introduce errors.
  - summation of more than 2 elements can be done quicker.

• Evaluate the effects of staging.

• Allow better control over the FP arithmetic in SML.
  - processor flags
  - precision of internal FP registers

• Design a language that can compile and *run* the predicates
  - run-time code generation and meta-programming
  - application: robust solid modeler
Conclusions

• **Automated conversion** of exact predicates into floating point code:
  • preserves correctness of computation despite rounding
  • compile-time error analysis
  • nested anonymous functions (staging)
  • performance of generated predicates close to hand-generated code

• **Formal specification** of the compiler.

• **More control** needed over floating point in SML.
  - processor flags
  - precision of internal FP registers

• **Compiler** can be found at http://www.cs.cmu.edu/~pscico
Operations on Expansions (arbitrary precision arithmetic)

Example: Adding two expansions

\[ E = \{e_1, e_3, e_5\} \quad F = \{e_2, e_4\} \]

\[ H = \{h_1, h_2, h_3, h_4, h_5\} \]
Source Language

• Basic arithmetic operations: +, -, £, sq, unary –

• Assignments: val x = some expression

• Nested anonymous functions (staging)
fn [ a, b, c ] =>
    let val ab = a + b
    val ab2 = sq ab
    val d = c - ab2
end

let val E = R.fromManExp{man=1.0, exp= ~53} in
    fn [ a, b, c ] =>
        let val ab = a + b
        val ab2 = ab * ab
        val d = c - ab2
        val dP = (abs c) + ab2
        val yAE = errA * dP
        in
                if d > yAE then 1
                else if ~ d > yAE then ~1 else

let val abB2 = Q.sq(ab)
    val dB = Q.isub(c, abB2)
    val yBX = X.approx(dB)
    val yBE = errB * ab2
    in
        if yBX > yBE then 1
        else if ~ yBX > yBE then ~1 else

let val abC = #err (Q.toResErr(Q.diff(a, b)))
    val abC2 = 2.0 * (ab * abC)
    val dC = ~ abC2
    val yCX = yBX + dC
    val yCE = errCX * yBX + errC * ab2
    in
        if yCX > yCE then 1
        else if ~ yCX > yCE then ~1 else

let val abD2 = Q.(Q.double(Q.prod(ab, abC)),
            Q.sq(abC))
    val dD = X.(~(abD2))
    in
        X.sign(X.(dB, dD))
end
end
end
end (* main let *)
Compiling a Program (cont’d)

```ml
fn [a, b, c] =>
  let val ab = a + b
  val ab2 = sq ab
  val d = c - ab2
end
```

```ml
let val E = R.fromManExp{man=1.0, exp= ~53}
  val errA = (3.0 + 10.0*E) * E
  val errB = (2.0 + 10.0*E) * E
  val errC = (3.0 + 13.0*E) * E * E
  val errCX = (2.0 + 7.0*E) * E
in
fn [a, b, c] =>
  let val ab = a + b
  val ab2 = ab * ab
  val d = c - ab2
  val dP = (abs c) + ab2
  val yAE = errA * dP
in
  if d > yAE then 1
  else if ~ d > yAE then ~1 else 0
end
```
Here goes example for Orient2D in SML or C

Nothing! They are just often mistaken for real numbers.

type pnt=real*real

fun orient(A:pnt, B:pnt, C:pnt)=
let val (a1, a2) = A
  val (b1, b2) = B
  val (c1, c2) = C
  val d = (c1-a1)*(c2-b2) - (c2-a2)*(c1-b1)
  in
    if d > 0 then Left else
    if d < 0 then Right else On
  end
What’s Wrong With Floating Point Numbers?

Floats are unevenly distributed on the real axis, thus introducing rounding errors in the computation.
What’s Wrong? (cont’d)

Example: Assume precision of 2 decimal digits. Denote © the operation of (rounded) addition on floats.

\[(5.0 + 5.3) + 1.3 = 11.6\]
\[(5.0 \oplus 5.3) \oplus 1.3 = 10. \oplus 1.3 = 11.\]

Floats are a *subset* but NOT a *subtype* of rationals and reals.
Types of Numbers

Real numbers
+ Usually assumed when developing or proving algorithms
- Basic operations not computable

Rational numbers
+ Closed under basic operations
- Slow

Floating-point numbers
+ Fast
  - Not closed under basic arithmetic operations (rounding errors)
  - Do not satisfy usual laws like associativity, distributivity
Phase A Compilation

- **Judgment** \( E_1 \vdash_A \alpha \leftrightarrow \lambda; r_1, r_2/E_2 \)

- **Selected rules**

\[
E_1 \vdash_A \text{val } x = e \leftrightarrow \lambda_H; s_1, s_2/E' \\
E' \vdash_A \alpha \leftrightarrow \lambda_T; r_1, r_2/E_2 \\
\hline
E_1 \vdash_A \text{val } x = e \alpha \leftrightarrow \lambda_H \lambda_T; r_1, r_2/E_2
\]

\[
E \vdash_A \text{val } y = x_1 + x_2 \leftrightarrow \\
\text{val } y^A = x_1^A \oplus x_2^A \quad \text{val } y^P = x_1^P \oplus x_2^P; \\
y^A, \left[ \frac{(1+\epsilon)^2}{1-\epsilon} \max(\delta_1, \delta_2) \right]_{fp} \otimes y^P/ \\
E, y^A : O_A(\epsilon + (1 + \epsilon) \max(\delta_1, \delta_2)), y^P : \mathcal{T}
\]
Phase B Compilation

• Judgment  \[ E_1 \vdash_B \alpha \mapsto \lambda; r_1, r_2/E_2 \]

• Selected rules

\[
\begin{align*}
E(x_1^A) = E(x_2^A) &= 0 \\
E &\vdash_B \text{val } y = x_1 + x_2 \mapsto \text{empty}; 0, 0/ \\
E, y^B : O_B(\epsilon), y^B &\doteq y^A
\end{align*}
\]
Operations on Expansions

Example: Adding a double to an expansion.
Overlapping and Machine $\varepsilon$

\[
\frac{x + \Delta}{a + b} = \frac{10101010...000.10}{10001010...01011}
\]

- $\Delta$ and $x$ do not overlap $\iff \Delta \cdot 2^{-p} x$.
- In IEEE double precision, $p=53$. 

42
Phases of Increasing Precision

let R = F (X) in floating point
if abs(R) > \textbf{estimated error bound}
then sign(R) else
\textbf{increase precision} and repeat
or switch to exact arithmetic

\hspace{\textwidth}

\textbullet Finitely many intermediate phases!
Arithmetic

Rational numbers
+ exact
- slow

Floating-point numbers
- inexact
+ fast
Non-overlapping

- The rounded sum $x$ and its rounding error $\Delta$ do not overlap.

\[
\begin{array}{c}
x \\
+ \Delta \\
\hline
x + \Delta
\end{array}
\]

\[
\begin{array}{c}
10101010...000.10 \\
0000............0.00 \\
10001010...01011
\end{array}
\]

\[
\begin{array}{c}
10101010...000,10.....10001010...01011
\end{array}
\]

- Mathematically, $|\Delta| \leq 2^{-p}|x| = \epsilon|x|$
Expansions and Compiler

Exact expression F

“filters out” the easy cases

compilation

let R = F(X) in floating point
if R > error bound then 1
else if –R > error bound then –1 else
increase precision and repeat
or switch to exact arithmetic

Use expansions for more precise computations
Phases and Compiler

let R = F(X) in floating point
  if R > error bound then 1
  else if –R > error bound then –1 else

  R2 = phase B
  if abs(R2) > error bound
  then sign(R2) else

  R3 = phase C
  if abs(R3) > error bound
  then sign(R3) else

  R4 = phase D; (* exact phase *)
  sign (R4)
Bounding the Rounding Error

- Floating-point operations are correctly rounded.
- Consequence: for any operation $\notin 2 \{+,-,\times\}$

$$x \cdot y = x \odot y \pm \varepsilon |x \odot y|$$

- Notice: $\varepsilon$ is static part of the error while $|x - y|$ is dynamic.
Bounding Error of Composite Expressions

- If $X_1 = x_1 \mathcal{S} \delta_1 p_1$ and $X_2 = x_2 \mathcal{S} \delta_2 p_2$ then

$$x_1 \oplus x_2 \pm (\epsilon + \max(\delta_1, \delta_2)(1 + \epsilon)) \underbrace{(p_1 \oplus p_2)}_{\text{static}} \underbrace{(p_1 \oplus p_2)}_{\text{dynamic}}$$
Error Bounds and Compilation

• Static error bound
  - Obtained EXACTLY in compile-time.
  - Rounded and then emitted into the target program as a floating-point constant.

• Dynamic error bound
  - Code must be generated for its computation and emitted into the target program.
Error Bounds and Compilation (cont’d)

let R = F(X) in floating point
  \[dynamic\_error = D(X)\]
  \[error = static\_error \times dynamic\_error\]
if R > error then 1
else if −R > error then −1
else
  jump to phases B, C and D
Compilation example

• Source expression c – (a + b)^2

• Target Standard ML program:

```ml
let val E = R.fromManExp{man=1.0, exp= ~53}
  val errA = (3.0 + 10.0*E) * E
  val errB = (2.0 + 10.0*E) * E
  val errC = (3.0 + 13.0*E) * E * E
  val errCX = (2.0 + 7.0*E) * E
in
  fn [ a, b, c ] =>
    let val ab = a + b
    val ab2 = ab * ab
    val d = c - ab2
    val dP = (abs c) + ab2
    val yAE = errA * dP
    in
      if d > yAE then 1
        else if ~ d > yAE then ~1 else
    end
    let val abD2 = Q.+(Q.double(Q.prod(ab, abC)), Q.-(abD))
    in
      X.+(dB, dD)
    end

let val abC = #err (Q.toResErr(Q.diff(a, b)))
  val abC2 = 2.0 * (ab * abC)
  val dC = ~ abC2
  val yCX = yBX + dC
  val yCE = errCX * yBX + errC * ab2
in
  if yCX > yCE then 1
    else if ~ yCX > yCE then ~1 else
    let val abD2 = Q.+(Q.double(Q.prod(ab, abC)), Q.-(abD))
    in
      X.+(dB, dD)
    end
end
```
Source Language

- Basic arithmetic operations: +, -, £, sq, unary -
- Assignments: val x = some expression
- Nested anonymous functions

**phrases**

\[ \phi ::= x \mid c \mid e \]

**expressions**

\[ e ::= \phi_1 + \phi_2 \mid \phi_1 - \phi_2 \mid \phi_1 \times \phi_2 \mid -\phi \mid sq \phi \]

**assignment lists**

\[ \alpha ::= \text{val } x = e \mid \text{val } x = e \alpha \]

**programs**

\[ \pi ::= \text{fn } [x_1, \ldots, x_n] \Rightarrow \]

\[ \begin{align*}
\text{let } \alpha \text{ end} \\
\text{fn } [x_1, \ldots, x_n] \Rightarrow \\
\text{let } \alpha \pi \text{ end}
\end{align*} \]
Staging (cont’d)

(* staged 2D Orientation predicate *)

fn [ ax, ay, cx, cy ] =>
  let val acx = ax - cx
  val acy = ay - cy
  in
  fn [ bx, by ] =>
    let val d = acx * ( by - cy ) +
      acy * ( bx - cx )
    in
    sign d
  end
end

\[
\begin{array}{c|cc|}
\text{sign} & A_x - C_x & B_x - C_x \\ \hline
A_y - C_y & B_y - C_y
\end{array}
\]
Overview

• Introduction
• Arbitrary precision arithmetic and computing in phases
• Multi-stage filters
• Performance
• Conclusion and Future Work